Future Generation Computer Systems 83 (2018) 50–59

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Serverless computing for container-based architectures

Alfonso Pérez *, Germán Moltó, Miguel Caballer, Amanda Calatrava

Instituto de Instrumentación para Imagen Molecular (I3M), Centro mixto CSIC - Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia, Spain

HIGHLIGHTS

- A framework to run containerized applications in serverless computing is proposed.
- Containers out of Docker images can now be run on AWS Lambda.
- Highly-parallel event-driven file-processing serverless computing is introduced.
- An analysis of the Freeze/Thaw cycle of AWS Lambda and caching is assessed.
- Bursty workloads of short stateless jobs can benefit from serverless computing.

ARTICLE INFO

Article history:
Received 26 July 2017
Received in revised form 5 January 2018
Accepted 9 January 2018

Keywords:
Cloud computing
Serverless
Docker
Elasticity
AWS lambda

ABSTRACT

New architectural patterns (e.g. microservices), the massive adoption of Linux containers (e.g. Docker containers), and improvements in key features of Cloud computing such as auto-scaling, have helped developers to decouple complex and monolithic systems into smaller stateless services. In turn, Cloud providers have introduced *serverless* computing, where applications can be defined as a workflow of event-triggered functions. However, serverless services, such as AWS Lambda, impose serious restrictions for these applications (e.g. using a predefined set of programming languages or difficulting the installation and deployment of external libraries). This paper addresses such issues by introducing a framework and a methodology to create Serverless Container-aware ARchitectures (SCAR). The SCAR framework can be used to create highly-parallel event-driven serverless applications that run on customized runtime environments defined as Docker images on top of AWS Lambda. This paper describes the architecture of SCAR together with the cache-based optimizations applied to minimize cost, exemplified on a massive image processing use case. The results show that, by means of SCAR, AWS Lambda becomes a convenient platform for High Throughput Computing, specially for highly-parallel bursty workloads of short stateless jobs.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Cloud computing introduced the ability to provision on-demand computational resources reducing the needs for on-premises resources. Indeed, Virtual Machines (VMs) have played a fundamental role to create customized and replicable execution environments for applications, in order to guarantee successful executions. Also, elasticity has been the cornerstone functionality of IaaS (*Infrastructure as a Service*) Cloud computing where new VMs can be provisioned in order to cope with increased workloads. Public Cloud providers such as Amazon Web Services (AWS) [1] have fostered the migration of complex application architectures to the Cloud in order to take advantage of the pay-per-use cost model.

In parallel, the mainstream adoption of Linux containers, propelled by the popularity of Docker [2], enabled users to maintain customized execution environments, in the shape of lightweight Docker images instead of bulky Virtual Machine Images. This paved the way for the microservices architectural pattern to rise, in order to decouple complex applications into several small, independently deployed services that interact via REST interfaces [3]. Creating distributed applications based on microservices required the ability to manage a fleet of Docker container at scale, thus fostering the appearance of Container Management Platforms (CMPs) such as Kubernetes, Apache Mesos or Docker Swarm. Public Cloud providers also provided their CMP offerings as a service, as is the case of Amazon ECS [4].

The ability to run containers at scale was adopted by public Cloud providers to create *serverless* computing [5] in which applications are defined as a set of event-triggered functions that execute without requiring the user to explicitly manage servers. As

* Corresponding author.
E-mail address: alpegon3@upv.es (A. Pérez).

https://doi.org/10.1016/j.future.2018.01.022
0167-739X/© 2018 Elsevier B.V. All rights reserved.

an example, AWS Lambda supports functions defined in different programming languages. AWS Lambda executes those functions on specific runtime environments provided by containers specifically tailored for the execution of the function, depending on the language chosen.

Serverless computing introduces large-scale parallelism and it was specifically designed for event-driven applications that require to carry out lightweight processing in response to an event (e.g. to do a minor image manipulation when a file is uploaded to an Amazon S3 bucket or to access a NoSQL back-end to retrieve data upon invocation of a REST API). However, the programming model that these services impose, as is the case AWS Lambda, hinder the adoption of this service for the general execution of applications. It is true that one can include a binary application in a deployment package for AWS Lambda but, still, managing the execution of generic applications in AWS Lambda is far from being a trivial task. Considering the popularity of Docker as a software distribution approach, it would be convenient to be able to run containers out of Docker images in Docker Hub on AWS Lambda to support generic execution of applications on such a serverless platform.

To this aim, this paper introduces SCAR, a framework to transparently execute containers (e.g. Docker) in serverless platforms (e.g. AWS Lambda). The following benefits are obtained: First, the ability to run customized execution environments in such platforms opens up new approaches to adopt serverless computing in scientific scenarios that were previously unable to easily exploit the benefits of such computing platforms. Second, new programming languages can be used apart from those natively supported by the serverless platform. Third, a highly-parallel event-driven file-processing serverless execution model is defined. This has allowed to run on AWS Lambda High Throughput Computing applications on customized runtime environments provided by Docker containers.

After the introduction, the remainder of the paper is structured as follows. First, Section 2 describes the related work in this area. Then, Section 3 describes the SCAR framework together with the underlying technology employed and the programming model that it introduces for event-driven file-processing applications. Next, Section 4 describes different use cases of this framework together with execution results in order to evaluate the benefits and limitations of the framework. Finally, Section 5 summarizes the main achievements and points to future work.

2. Related work

Serverless computing is a new execution model that is currently emerging to transform the design and development of modern scalable applications. Its evolution is reinforced by the continuous advances in container-based technology together with the consolidation of cloud computing platforms. In this way, new event-driven services have appeared in the last three years. AWS Lambda [6] was the first serverless computing service to appear offered by Amazon Web Services, followed by Google Cloud Functions [7], Microsoft Azure Functions [8], and the open source platform Apache OpenWhisk [9]. A discussion about all these services can be found in the work by McGrath et al. [10], where authors evaluate the usage of this new event-driven technology with two different case studies, outlining the potential of cloud event-based services.

Although it is a relatively new area, there exists in the literature several works contributing to the evolution of serverless computing. For example, the initial developments of OpenLambda are presented in [11] as an open-source platform for building web services applications with the model of serverless computing. The work also includes a case study where performance of executions in AWS Lambda are compared with executions in AWS Elastic Beanstalk [12], which concludes with better performance results for AWS Lambda. The study by Villamizar et al. [13] presents a cost comparison of a web application developed and deployed using three different approaches: a monolithic architecture, a microservices architecture operated by the cloud customer, and a microservices architecture operated by AWS Lambda. Results show that AWS Lambda reduces infrastructure costs more than 70% and guarantees the same performance and response times as the number of users increases.

Several tools related with serverless computing are emerging in the literature. First, Podilizer [14] is a tool that implements the pipeline specifically for Java source code as input and AWS Lambda as output. Second, Snafu [15] is a modular system to host, execute and manage language-level functions offered as stateless microservices to diverse external triggers. Finally, [16] presents the prototype implementation of PyWren, a seamless map primitive from Python on top of AWS Lambda that is able to reuse one registered Lambda function to execute different user-defined Python functions.

Also, use cases of this emerging event-based programming model can be found in the literature, like the work by Yan et al. [17] where the authors present a prototype architecture of a chatbot using the OpenWhisk platform, or the experiments described in [18] about face recognition with LEON, a research prototype built with OpenWhisk, Node-RED [19] and Docker. Another use of serverless computing is data analytics, exemplified in [20], covering data processing with Spark and OpenWhisk.

All the works mentioned above highlight the advantages of using these new event-driven services because they are more elastic and scalable than previous platforms. Moreover, they point out the challenges derived from the granular nature of serverless computing. The work by Baldini et al. [5] regarding the open problems of serverless computing identifies several unaddressed challenges which include: (i) the ability to run legacy code on serverless platforms, and (ii) the lack of patterns for building serverless solutions.

Indeed, this paper addresses the aforementioned challenges by introducing an open-source framework that enables users to run generic applications on a serverless platform, AWS Lambda in this case. This introduces unprecedented flexibility for the adoption of serverless computing for different kind of applications. Previously, users were constrained to create functions on the specific programming languages supported by AWS Lambda. In addition, we introduce a High Throughput Computing programming model to create highly-parallel event-driven file-processing serverless applications that execute on customized runtime environments (provided by Docker images) on AWS Lambda.

Therefore, the main scientific contribution of SCAR is to democratize serverless computing for a wide range of scientific applications and programming languages that were not previously able to easily exploit a serverless platform. As far as the authors' knowledge, this is the first work in the literature that proposes, and provides an open-source implementation, a framework to run containers out of Docker images in a serverless platform such as AWS Lambda. Indeed, several examples of applications successfully ported to serverless computing on AWS Lambda with SCAR are already available in the GitHub repository [21]. These include, but are not limited to, deep learning frameworks, such as Theano [22] and Darknet [23], programming languages such as Erlang [24] and Elixir [25] and generic tools for image and video processing such as ImageMagick [26] and FFmpeg [27]. We believe that the integration of Docker for application delivery with a serverless platform such as AWS Lambda provides an appropriate platform for different computing scenarios that require fast elasticity and the ability to scale beyond the limits of current IaaS Clouds.

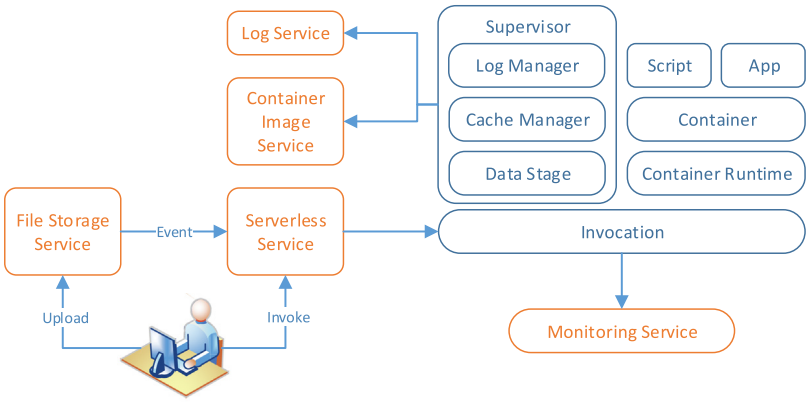

Fig. 1. Architectural approach for supporting container-based applications on serverless platforms.

3. The SCAR framework

Fig. 1 describes the architectural approach designed to support container-based applications on a serverless platform. The services typically made available by the Cloud provider are: (i) *Serverless service*, also known as Functions as a Service (FaaS), responsible for executing the Cloud functions in response to events; (ii) *File storage service*, which hosts the files uploaded by the user and triggers the events to the Serverless service so that the file can be processed by the invocation of the function. *N* file uploads will trigger *N* invocations of the function where each one processes exactly one file; (iii) *Log service*, where the information concerning the execution of the function is logged; (iv) *Monitoring service*, which provides metrics of the resources consumed by the function. In addition, a *Container Image Service* is required, in order to store the images that include the operating system together with the application and its dependencies.

An invocation of the function involves the execution of the *Supervisor*, responsible for: (i) *Data stage* from the File storage service into the temporary data space allocated to that particular function invocation; (ii) *Cache management* in order to minimize the data movement from the Container image service to the data space available to the function; (iii) *Log management*, to retrieve the output of the execution of the container. The supervisor delegates on a *Container runtime* in order to instantiate a *Container* out of an image, on which either a script or an application is run on the customized runtime environment provided by the container.

3.1. Underlying technology employed in SCAR

The following subsection identifies and justifies the different technology choices made to develop SCAR.

3.1.1. Cloud provider services: AWS

As described in the related work section, there are different services for serverless computing. We chose AWS Lambda [6], a serverless computing service to run code, in the shape of functions created in a programming language supported, in response to events so that no explicit management of servers is performed by the user. The most important features and limitations of AWS Lambda are: (i) Constrained computing capacity currently limited by a maximum of 3008 MB, where CPU performance is correlated with the amount of the memory chosen; (ii) Maximum execution time of 300 s (5 min); (iii) Read-only file system based on Amazon Linux; (iv) 512 MB of disk space in */tmp*, which *may* be shared across different invocations of the same Lambda function; (v) Default concurrent execution limit of 1000 invocations of the same function, which can be increased, (vi) Supported execution environments: Node.js v4 and 6, Java 8, Python 3.6 and 2.7, .NET Core 1.01 (C#), and (vii) No inbound connections are allowed for the Lambda invocations.

For the file storage service, Amazon S3 (Simple Storage Service) was chosen, an object storage designed to provide durable and highly available access to files, stored in *buckets*, which are created in a specific AWS Region. S3 can publish event notifications when certain actions occur. For example, the *s3:ObjectCreated* event type is published whenever the S3 APIs such as PUT, POST or COPY are used to create an object in a bucket. These events can be published for different destinations (services) such as Amazon SNS (a push messaging service), Amazon SQS (a message queuing service) and AWS Lambda.

Amazon CloudWatch [28] is the monitoring service of AWS. In particular, CloudWatch Logs is a service to monitor, store and access log files produced from different sources and services in AWS. Therefore, the standard output generated by the Lambda function invocations are sent to CloudWatch Logs so that different Log streams are obtained from which to obtain the information regarding the execution of a given invocation.

3.1.2. Containers: Docker, Docker Hub and udocker

Among the different choices for Linux containers, such as OpenVZ [29] and LXC/LXD [30], we chose Docker due to its mainstream adoption for software delivery. This is exemplified by Docker Hub [31] a cloud-based registry service that hosts Docker images and can automatically create them by linking code repositories, thus providing a centralized place to distribute Docker images.

Since external packages cannot be installed on a FaaS platform, i.e., no root privileges are available to install Docker, a mechanism to run a container out of a Docker image on user space without requiring prior installation is needed. This is precisely the ability of udocker [32,33], a tool to execute containers in user space out of Docker images without requiring root privileges. This allows pulling images from Docker Hub and create containers by non-privileged users on systems where Docker cannot be installed. This tool has demonstrated to be useful to run jobs on customized execution environments in both Grid environments (such as the European Grid Infrastructure) and HPC (High Performance Computing) clusters of PCs in the context of the INDIGO-DataCloud European project [34].

Udocker provides several execution modes, described in the documentation [35]. However, due to the restrictions of the execution environment provided in AWS Lambda, only the *F*1 execution mode properly works, which involves using Fakechroot [36] with direct loader invocation. Using this approach, it is possible to run a process in the execution environment provided by a Docker image without actually creating a Docker container.

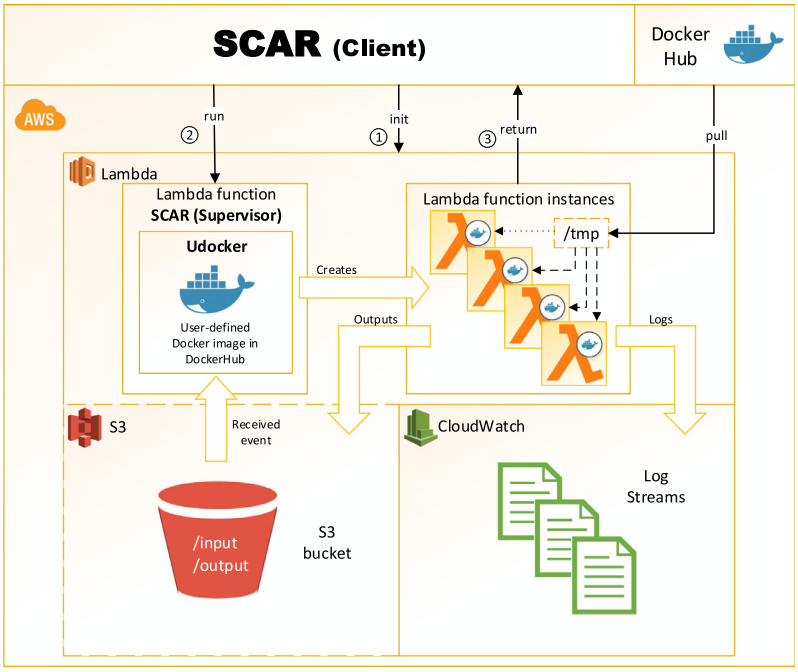

Fig. 2. Architecture of SCAR.

Notice that process isolation is automatically provided by the execution model of AWS Lambda, where different invocations of the same function are run on isolated runtime spaces.

3.2. Architecture of SCAR

SCAR allows users to define Lambda functions, where each invocation will be responsible for executing a container from a Docker image stored in Docker Hub and, optionally, execute a shell-script inside the container for further versatility. Fig. 2 describes the architecture of SCAR. The framework architecture is divided in two parts:

- *SCAR Client*. The client is a Python script that provides a Command-Line Interface (CLI) that is responsible for: (i) validating the input information from the user; (ii) creating the deployment package, which includes udocker; (iii) creating the Lambda function containing the SCAR Supervisor; (iv) providing an easy access to the Logs generated by each invocation of the Lambda function; (v) providing means for the user to manage the lifecycle of the Lambda function (init, list, run, delete) and (vi) manage the configuration to trigger events from an S3 bucket to a Lambda function. The client heavily uses the Boto 3 library [37] to interact with the AWS services
- *SCAR Supervisor*. The supervisor represents the code of the Lambda function, which targets the Python 3.6 runtime environment and is responsible for: (i) retrieving the Docker image from Docker Hub using udocker into */tmp*, unless the Docker image is already available there; (ii) creating the container out of the Docker image and setting the appropriate execution mode for udocker; (iii) in case of being triggered from S3, manage the staging of the input data into the container and the stage out of the output results back into S3; (iv) passing down the environment variables to the container (those defined by the user and others of interest, such as the temporary credentials, so that the code running in the container has precisely the same privileges as the Lambda function itself; (v) merging the generated output from the script running in the container to the Lambda output in order to have consolidated logging available in CloudWatch Logs.

The usage of SCAR in order to run a generic application on AWS Lambda is as follows, as described in Fig. 2. First, the user chooses a Docker image available in Docker Hub and creates (*init*) the Lambda function with the specific performance configuration provided by the user (in terms of memory). Second, the user can directly invoke (*run*) the Lambda function. This triggers the SCAR supervisor which, as described earlier, effectively ends up executing a container out of a Docker image and optionally run a user-defined shell script. Data staging from and to S3 is automatically managed by the SCAR supervisor, together with diverting the logs into CloudWatch.

Notice that *caching* is a fundamental technique for the SCAR framework, especially considering the maximum execution time of 300 s. The first invocation of a Lambda function will pull the Docker image from Docker Hub into */tmp*, which can take a considerable amount of time (in the order of seconds), depending on the size of the Docker image. Subsequent invocations of the Lambda function *may* already find that Docker image available in */tmp* and, therefore, there is no need to retrieve the Docker image again from Docker Hub.

However, caching does not restrict to the Docker image. In addition, the container created with udocker, is also shared among all the Lambda invocations that may find it already available in */tmp*. The rationale behind this approach is that since Lambda functions are provided with a read-only file system, so are provided the scripts executed in the containers run on the Lambda functions. Notice that due to the stateless environment inherent to the Lambda functions, caching does not introduce side effects. It just reduces the invocation time whenever the cache is hit and the container is already available.

Therefore, the duration of the Lambda function invocation using SCAR is greatly dependent on the ability for the Lambda functions to find a cached container file system in */tmp*. This will be thoroughly assessed in Section 4.

```
OUTPUT_DIR="/tmp/$REQUEST_ID/output"

echo "SCRIPT: Invoked Video Grayifier. File available in $SCAR_INPUT_FILE"
FILENAME=‘basename $SCAR_INPUT_FILE‘
OUTPUT_FILE=$OUTPUT_DIR/$FILENAME
echo "SCRIPT: Converting input video file $SCAR_INPUT_FILE to grayscale to output file $OUTPUT_FILE"
ffmpeg -loglevel panic -nostats -i $SCAR_INPUT_FILE -vf format=gray $OUTPUT_FILE < /dev/null
```

Fig. 3. Sample script to perform video transcoding.

Concerning the overhead introduced by SCAR, users should be aware that it requires a reduced amount of memory and disk space to run (∼36 MB of RAM and ∼16 MB of disk space). Indeed, it is the size of the container what really determines how much disk space is going to be available to be used by the applications. Empirical experimentations show that an image in Docker Hub larger than 220 MB will hardly fit inside the ephemeral storage allocated to the Lambda function, due to the storage requirements of both the Docker image and the container file system unpacked by udocker. Regarding the remaining time available for the execution of the application, once the first invocation is finished and the container is cached, SCAR takes a negligible time to check if the container exists, thus allowing the application to run during almost the maximum time provided by the function.

3.3. Event-Driven File-Processing serverless programming model

SCAR introduces a programming model that can be effectively used to process files on demand, by triggering the execution of the Lambda function as soon as a file is available in an Amazon S3 bucket. This is a summary of the programming model, exemplified by an application to transform videos into a grayscale using the well-known *ffmpeg* application. This use case is available online for the reader to test it [38].

The following command creates a Lambda function that will be triggered for any video upload to the *input* folder in the *scar-test* bucket. Upon invocation of the Lambda function, a container out of the *sameersbn/ffmpeg* Docker image (available in Docker Hub) will be started and the *process.sh* shell-script will be executed inside.

```
scar init -s process.sh -n lambda-ffmpeg -es
scar-test sameersbn/ffmpeg
```

The programming model results in the following workflow:

1. The user uploads the video into the `input` folder of the `scar-test` S3 bucket.
2. The input video file is made available to the executed container in `/tmp/$REQUEST_ID/input`, as specified by the `$SCAR_INPUT_FILE` environment variable.
3. The script converts the video to grayscale and saves the output file into `/tmp/$REQUEST_ID/output`.
4. The video file is automatically uploaded to the `output` folder of the Amazon S3 bucket and deleted from the underlying storage.

The content of the script is included in Fig. 3. Notice the simplicity of the script that just invokes the command-line application in order to process the video using FFmpeg into an output folder. Since data stage is automatically managed by the SCAR Supervisor (running on the Lambda function), the user just focuses on how to specifically process one file. Then, multiple instances of this script can be run in parallel, each one running on its own Lambda invocation in order to simultaneously process different videos at scale, taking into account the limited storage space available. This approach also facilitates testing, which can be performed locally to process one file within a Docker container and then can be scaled out to thousands of concurrent invocations run on AWS Lambda.

It is important to point out that there is currently no other work in the literature that proposes a High Throughput Computing Programming Model to create highly-parallel event-driven file-processing serverless applications that execute on customized runtime environments provided by Docker containers run on a serverless platform such as AWS Lambda.

4. Use cases: Results and discussion

This section provides a thorough assessment of the capabilities and limitations of SCAR, which are greatly dependent on the underlying features of AWS Lambda. As stated in Section 3.1.1, Lambda functions have a maximum execution time of 300 s and an allocated temporary space (in */tmp*) of 512 MB. Going beyond those thresholds make the Lambda function invocation fail.

According to the documentation [39], when a Lambda function is invoked, AWS Lambda launches a container (i.e., an execution environment). After a Lambda function is executed, the service keeps the container for some time just in case the same Lambda function is invoked again. The service *freezes* the container after a Lambda function finishes, and *thaws* the container for reuse, in the case AWS Lambda decides to reuse the container when the Lambda function is invoked again, a process known as the freeze/thaw cycle. Therefore the */tmp* content remains when the container is frozen, providing a transient cache that can be used for multiple invocations

Notice that, when using SCAR, a container in user-space is run via udocker inside the container provided by the AWS Lambda invocation. Being able to cache the user-space container dramatically affects the execution time of a Lambda function created by SCAR. To this end, Section 4.1 provides a comprehensive study of the reuse of the ephemeral disk space. Next, Section 4.2 introduces a realistic use case of the programming model presented in Section 3.3: A customized execution environment containing an open-source deep learning framework is used to recognize and classify a set of images stored in a Cloud provider making use of the massive scaling capabilities of the serverless architectures with the help of SCAR.

4.1. On the Freeze/Thaw cycle: Disk space reuse

Due to the importance of the AWS Lambda Freeze/Thaw cycle for the SCAR framework, an study of the behavior of this feature is conducted in order to extract optimized invocation patterns to be used in SCAR. For all the tests in this study, only the time used to download the Docker image from Docker Hub and run actions (i.e., creation of the container via udocker) are measured. The time spent by the script executed inside the container is negligible for this study. Also, the memory was set to 512 MB for all the functions.

The first comparison is done between the two different invocation types that AWS Lambda offers (i.e. *request–response* and *asynchronous*) [40]. We analyze the time spent for each invocation. Each one involves creating a container out of a Docker image (in this case centos:7 [41]) stored in Docker Hub and executing a trivial shell-script inside. Ten different Lambda functions were created for each invocation type (i.e. a total of 20 different Lambda functions). Each function was invoked a hundred times (i.e. a total of 2.000

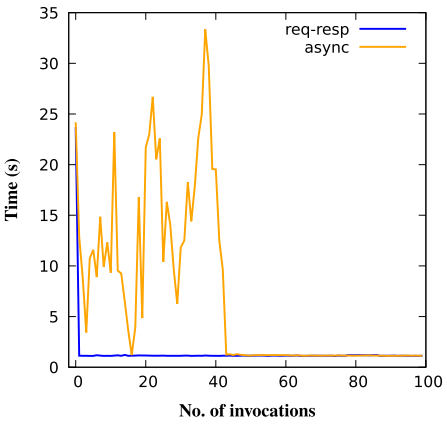

Fig. 4. Average execution time (in seconds) for each invocation type (i.e. request–response and asynchronous).

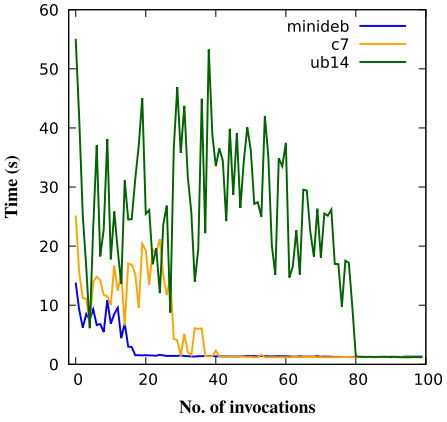

Fig. 5. Average execution time (in seconds) for different container sizes. All the invocations are asynchronous.

invocations). Fig. 4 shows the average execution time for each invocation type.

The *req–resp* line refers to the *request–response* invocation type and it shows that all the invocations performed after the first one take a negligible time to execute. This means that the first invocation spends 25 s on average downloading the container image from Docker Hub, creating the container and running a script inside it. However, the subsequent invocations execute very rapidly. This is a coherent behavior when using the request–response type, where subsequent invocations after the first one will find the container file system already available in the ephemeral space of the Lambda function invocation (i.e. cached by SCAR).

The *async* line refers to the *asynchronous* invocation type and it shows an slightly erratic behavior in the execution time of the functions along the first 40 invocations. Indeed, the asynchronous model carries out all the invocations almost simultaneously, though there is a slight delay due to the SCAR invocation manager. Therefore, the Lambda functions that are invoked first do not find the container cached in */tmp* and need to retrieve the Docker image from Docker Hub and create the container. This means that the execution is performed successfully, but requires additional time. After approximately 40 invocations, the container is finally cached in the ephemeral disk space and, therefore, the execution time of the subsequent invocations decreases considerably, as expected.

Therefore, the main conclusion of this first experiment is that a newly created Lambda function with SCAR should be executed at least once, in order to fully cache the container in the ephemeral disk space, before attempting to perform multiple asynchronous invocations.

To further investigate how the container size affects the caching behavior when using the asynchronous invocation type, we launched different functions that use different container image sizes and we analyzed the duration of the invocations. To carry out this experiment we created three different function types: the *minideb* type, which uses the Docker image *bitnami/minideb* [42] and has an image size of 22 MB; the *c7* type, which uses the Docker image *centos:7* [41] and has an image size of 70 MB; and finally the *ub14* type, which uses the Docker image *grycap/jenkins:ubuntu14.04-python* [43] and has an image size of 153 MB. Each one of these types are used to create ten functions (i.e. a total of 30 different Lambda functions) and each different function is invoked a hundred times (i.e. 3.000 invocations in total). For the sake of clarity, the execution times of the functions belonging to the same type are presented as average values. Fig. 5 shows the results of these invocations.

The *minideb* function, which represents the smallest container image, is the first to present a cached behavior. The *c7* function, which has the medium size container image, is cached after approximately 30 invocations and the function with the largest container image, *ub14*, requires more invocations, approximately 80, before it is cached. In this figure, it can be clearly seen the relation between increasing the container image size and the time that takes the container image to be cached in the ephemeral disk space. As explained above, this is directly related to the asynchronous way of working of the Lambda functions, in which the system does not wait for the previous invocation to finish. Therefore, the container image can be cached or not depending on the time passed since the first execution and the size of such container image.

As seen in the experiments, on the one hand there is the request–response invocation, where SCAR achieves a cached behavior starting from the second invocation, at the expense of waiting for that first invocation to end. On the other hand, there is the asynchronous invocation type, in which all the invocations can be carried out in parallel, but the container size affects the time until the containers are cached by SCAR in the ephemeral disk space.

In the programming model proposed by SCAR, the executions benefit from both invocation types by performing the first invocation as request–response and the rest invocations as asynchronous. To assess the advantages of this approach, an experiment was carried out using the aforementioned approach. The same function types described in the previous test were used to carry out this experiment: *minideb*, *c7* and *ub14*. These types were used to generate 30 different functions and each function was invoked a hundred times. Again, for the sake of clarity, the execution times of the functions belonging to the same type are presented as average values.

Fig. 6 presents the results of this experiment. The erratic behavior of the caching system has disappeared and all the invocations present a cached performance starting from the second invocation and thereafter. This approach allows SCAR to reduce the overall execution time and, therefore, we adopt it for the event-driven file processing programming model. As such, a Lambda function created with SCAR is previously *preheated*, i.e., invoked with a request–response type, so that subsequent parallel asynchronous invocations already find the container cached in the ephemeral disk space.

The last experiment designed for this case study investigates the influence of the time between invocations on the ability for AWS Lambda to reuse the container, which ends up on SCAR being able to find a cached container, thus speeding up the Lambda function invocation. Therefore, since the benefits of *preheating* a Lambda function were identified, it is important to know the time

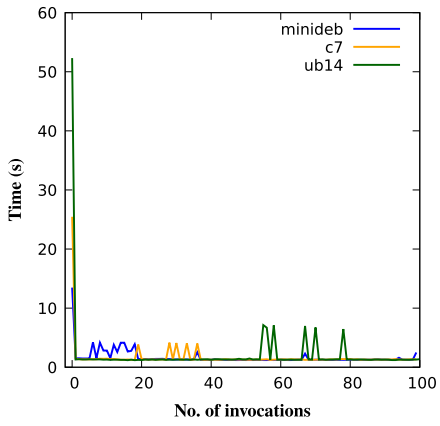

Fig. 6. Average execution time (in seconds) for different container sizes. The first invocation type is request–response and subsequent are asynchronous.

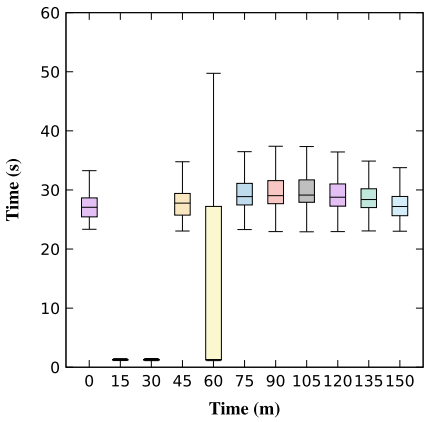

Fig. 7. Execution time (in seconds) used by the function related to the time waited (in minutes) from a previous invocation of the same Lambda function.

before it *cools down* again, i.e., when the invocation of the function will not reuse the same Lambda container and, therefore, SCAR will have to retrieve the Docker image again from Docker Hub and create a new user-space container.

To this end, this experiment invokes the Lambda function in predefined ranges of time (each one of them defined by increasing 15 min the previous waiting time). Eleven time ranges were defined, from 0 to 150 min of waiting time. A thousand different functions were created and invoked just once in each period, resulting in a total of 11.000 invocations launched.

Fig. 7 shows the results of this experiment using a box plot representation of the time used by each invocation. The whiskers of the box plot extend for a range equal to 1.5 times the interquartile range. By analyzing the figure we can extract three conclusions: first, all the functions show that the invocations are being cached between 15 and 30 min; second, if we wait more than 60 min the cache is lost and all of our functions need to download again the container image, and finally, waiting 45 or 60 min do not ensure the correct working of the cache due to the randomness of the underlying system.

Understanding the behavior of the Freeze/Thaw cycle and, therefore, when an invocation of a Lambda function created with SCAR will be cached, enables to adopt best practices when adopting serverless computing to execute generic applications. This knowledge is applied in the following section with a real example on image recognition using a deep learning framework.

4.2. Massive image processing

In this section SCAR is used to deploy a customized execution environment in order to recognize different patterns in images using deep learning techniques. The framework used to recognize the patterns is Darknet [44], an open source neural network framework written in C, in combination with the You Only Look Once (YOLO) [45] library. The Docker image used for this case study can be found in the *grycap/darknet* [46] Docker Hub repository and the memory set for the function in this case study is 1024 MB, which is the minimum function size that allows to run the experiment due to memory and time constraints.

The programming model presented in Section 3.3 has been extended with the ability to automatically perform Lambda invocations out a set of files already available in a Cloud storage service (Amazon S3, in this case). This feature allows the user to reuse an existing S3 bucket with data files in order to perform a High Throughput Computing analysis across all the files in that bucket. A Lambda invocation per file will be carried out (up to the 1.000 soft limit of concurrent Lambda invocations). Each Lambda invocation will execute a shell-script to process exactly one file.

For this experiment we use: (i) a thousand images of animals and objects already stored in an AWS S3 bucket with a size of almost 500 MB; (ii) the Docker image stored in Docker Hub which contains all the libraries and dependencies needed to execute the Darknet software and (iii) a shell-script executed inside the container, in charge of processing the input image, using DarkNet and the YOLO library to obtain the output (the patterns recognized in the image).

The script used to launch the Darknet object detection software is presented in Fig. 8. The SCAR_INPUT_FILE variable is created by SCAR to simplify the creation of the script and contains the name of the file received by the Lambda function invocation. Some output variables are created and the information about the script execution is written in the standard output. When the container execution finishes and the output files have been processed, SCAR writes all the standard output produced by the container in the Lambda function logs, thus easing the traceability of possible errors. The Darknet invocation command receives an image as an input file and stores the results in two separate files, the OUTPUT_IMAGE which will be the image with the recognized objects and the RESULT file which will contain the percentage of certainty for each recognized object. Sample input/output images are presented in Figs. 9 and 10 respectively. Also the output file generated after processing the image is shown in Fig. 11. To finish, the image produced is moved next to the output file into the output folder, so that SCAR can automatically upload them into S3. Remember that the user does not have to write any code to manage the download and upload of the input and output files respectively. All the files are managed transparently by SCAR.

After the execution of the case study, the following metrics were retrieved: 2 min used to finish the experiment; 880 min as the total aggregated execution time across the multiple Lambda function invocations; 4.575 different objects and animals recognized. As summary, in only two minutes we have downloaded, processed and uploaded a thousand images without worrying about the deployment and the management of the architecture.

It is important to point out four main conclusions that arise from this case study. Firstly, without SCAR the user has no easy way of using specific libraries such as Darknet in serverless providers like AWS Lambda. Secondly, the user does not have to manage the deployment of computational infrastructure, auto-scaling, coordinating the execution of jobs, etc. Instead, the serverless computing platform introduced seamless elasticity by performing multiple concurrent executions. Thirdly, the simplicity of the programming

```
OUTPUT_DIR="/tmp/$REQUEST_ID/output"
FILENAME=‘basename $SCAR_INPUT_FILE .jpg‘
RESULT="$OUTPUT_DIR/$FILENAME.out"
OUTPUT_IMAGE=$FILENAME-out

echo "SCRIPT: Received file ’$SCAR_INPUT_FILE’."
echo "SCRIPT: Saving result in ’$RESULT’ and output image in ’$OUTPUT_IMAGE.png’"

cd /opt/darknet
./darknet detect cfg/yolo.cfg yolo.weights $SCAR_INPUT_FILE -out $OUTPUT_IMAGE > $RESULT

mv $OUTPUT_IMAGE.png $OUTPUT_DIR/
```

Fig. 8. Script used to launch the YOLO object detection library of the Darknet framework. It also processes the inputs and outputs of the container.

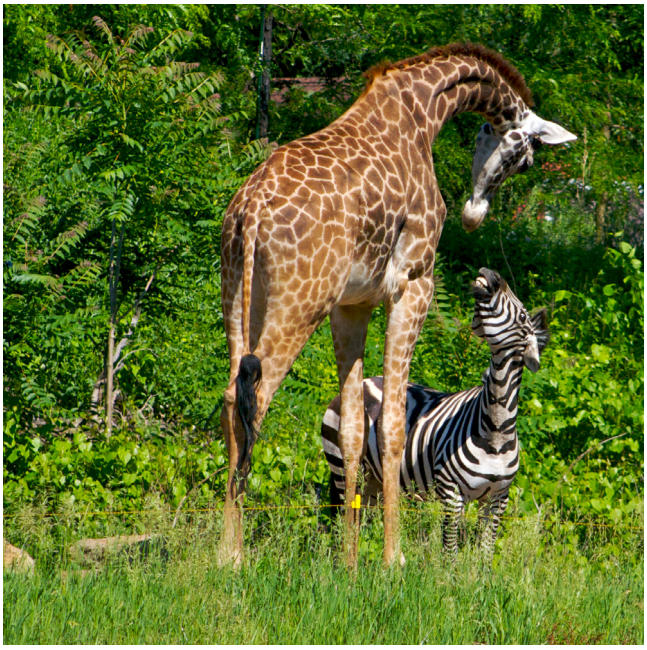

Fig. 9. Test image passed to the Darknet framework.

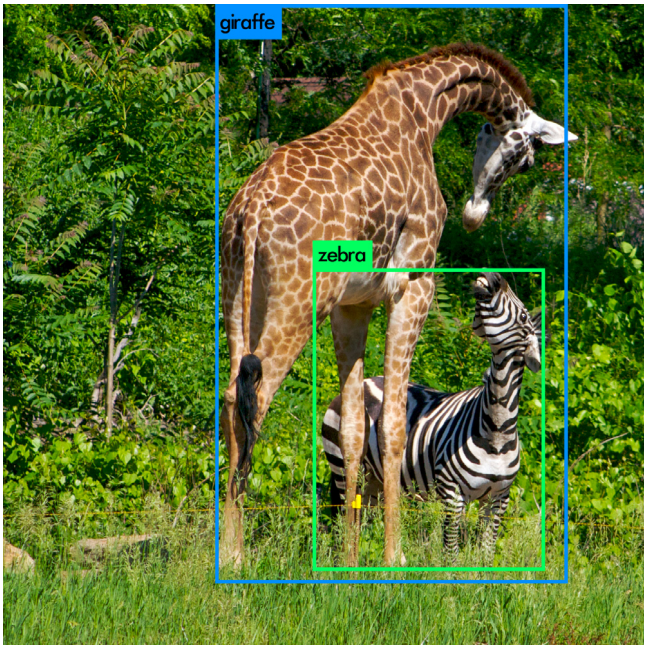

Fig. 10. Animals recognized after the execution of the YOLO library. This output image has been generated by the Darknet framework.

model introduced by SCAR just requires the user to write a shell-script to process a file assuming that will be automatically delivered. This is probably the simplest, most convenient approach to perform a file-processing application on the Cloud. Fourthly, once the Lambda function has been created by SCAR, this turns into a reactive service that is left on the Cloud service at no cost unless it is triggered again by uploading a new file to the bucket. This will cause a new Lambda function invocation, resulting in the creation of the container and execution of the shell-script to process the file. This has an important economic factor, since real pay-per-use is enforced as opposed to the pay-per-deploy approach that happens when deploying a VM in a Cloud service, which has a cost regardless of the actual usage of the VM.

Finally, notice that the ability to scale in the order of thousands of Lambda function invocations reduces the requirement for a job scheduler, in cases where the incoming workload can be seamlessly absorbed by the underlying computing platform.

To finish, we present a cost of the case study execution. Based on the metrics extracted earlier, the average execution time of the invocations is 52.8 s. Each function used 1024 MB of memory, and one invocation per photo available in the bucket was carried out, 1.000 in total. The AWS Lambda pricing calculator [47] indicates a cost of $0.88. Since AWS Lambda offers a free usage tier that includes 1 million requests and 400.000 GB-s of compute time per month, and this use case involved 1.000 requests and 52.000 GB-s, the real cost of classifying the images was $0.

5. Conclusions and future work

This paper has introduced SCAR, a framework to execute container-based applications using serverless computing, exemplified using Docker as the technology for containers and AWS Lambda as the underlying serverless platform. This is a step forward contribution to the state of the art, implemented in an open-source framework, that opens new avenues for adopting serverless computing for a myriad of scientific applications distributed as Docker images.

```
/tmp/b472565a-62ee-11e7-b079-4ff49957d32f/input/giraffe.jpg: Predicted in 11.708465 seconds.
giraffe: 90%
zebra: 80%
```

Fig. 11. Darknet output file produced for the test image. It contains the total execution time and the objects recognized (in this case animals) with a percentage of certainty.

Using the proposed approach, customized execution environments can now be employed instead of being locked-in to programming functions in the programming languages supported by the serverless platform (in our case AWS Lambda). This has easily introduced the ability to run generic applications on specific runtime environments defined by Docker Images stored in Docker Hub, a functionality that is actually missing from the current serverless computing platforms.

A High Throughput Computing programming model has been developed in order to create highly-parallel event-driven file-processing serverless applications that execute on customized runtime environments provided by Docker containers run on AWS Lambda. This has been exemplified by using a deep learning application to perform pattern recognition on an image dataset.

SCAR not only provides means to deploy containers in AWS Lambda, it also manages the Lambda functions' lifecycle and eases the execution of the serverless workflow by applying optimizations without the need of user intervention, such as caching the container's underlying file system to minimize the execution time.

However, the current limitations of AWS Lambda in terms of maximum execution time (5 min), maximum allocated memory (3008 MB) and, most important, ephemeral disk capacity (512 MB), impose serious restrictions for the applications that can benefit from SCAR. Bursty workloads of short stateless jobs are specially appropriate to benefit from the ultra-elastic capabilities of AWS Lambda, both in terms of the amount of concurrent Lambda function invocations (in the order of thousands) and the rapid elasticity (in the order of few seconds). Having said that, we expect these limits to be risen in future updates of the service, which will be greatly help expand the adoption of SCAR for applications that cannot be encapsulated in a Docker image fitting in such scarce amount of computing and storage resources.

Future work of SCAR includes adapting the development to other serverless providers. In particular, our dependence on udocker, begin developed in Python, suggests using a provider supporting that language, such as Microsoft Azure Functions (Google Cloud Functions currently only supports Node.JS). Notice that the programming model of SCAR is agnostic to the provider. In addition, SCAR users could benefit from a mechanism that maintains the deployed Lambda functions 'hot', based on the knowledge extracted from the freeze/thaw cycle study by means of periodic invocations of the Lambda functions. Finally, we are currently researching on mechanisms to checkpoint applications so that new Lambda functions are spawn recursively in order to bypass the maximum execution time for iterative scientific applications.

Acknowledgments

The authors would like to thank the Spanish "Ministerio de Economía, Industria y Competitividad" for the project "BigCLOE" under grant reference TIN2016-79951-R. The authors would also like to thank Jorge Gomes from LIP for the development of the *udocker* tool.

References

[1] Amazon, Amazon Web Services (AWS). https://aws.amazon.com. (Online; accessed 25 July 2017).
[2] Docker, Docker. https://www.docker.com. (Online; accessed 25 July 2017).
[3] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Larisa Safina, Microservices: yesterday, today and tomorrow, CoRR (2016). abs/1606.04036.
[4] Amazon, Amazon EC2 Container Service. https://aws.amazon.com/ecs/. (Online; accessed 25 July 2017).
[5] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, Philippe Suter, Serverless computing: Current trends and open problems, Jun. 2017, pp. 1-20.
[6] Amazon, Amazon Lambda (AWS Lambda). https://aws.amazon.com/lambda/. (Online; accessed 25 July 2017).
[7] Google, Google Cloud Functions. https://cloud.google.com/functions/. (Online; accessed 25 July 2017).
[8] Microsoft. Microsoft Azure Functions. https://azure.microsoft.com/en-in/services/functions/. (Online; accessed 25 July 2017).
[9] The Apache Software Foundation. Apache Openwhisk. http://openwhisk.org/. (Online; accessed 25 July 2017).
[10] G. Mcgrath, J. Short, S. Ennis, B. Judson, P. Brenner, Cloud event programming paradigms: Applications and analysis, in: 2016 IEEE 9th International Conference on Cloud Computing, CLOUD, June 2016, pp. 400–406.
[11] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Serverless computation with openlambda, in: 8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 16, USENIX Association, Denver, CO, 2016.
[12] Amazon, AWS Elastic Beanstalk. https://aws.amazon.com/elasticbeanstalk. (Online; accessed 25 July 2017).
[13] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold. Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, Mery Lang, Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS lambda architectures, Serv. Oriented Comput. Appl. 11 (2) (2017) 233–247.
[14] Josef Spillner, Serhii Dorodko, Java code analysis and transformation into AWS lambda functions, CoRR (2017). abs/1702.05510.
[15] Josef Spillner, Snafu: Function-as-a-Service (FAAS) runtime design and implementation, CoRR (2017). abs/1703.07562.
[16] Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, Occupy the cloud: Distributed computing for the 99%, CoRR (2017). abs/1702.04024.
[17] Mengting Yan, Paul Castro, Perry Cheng, Vatche Ishakian, Building a chatbot with serverless computing, in: Proceedings of the 1st International Workshop on Mashups of Things and Apis, MOTA '16, ACM, New York, NY, USA, 2016, pp. 1–5.
[18] Alex Glikson, Stefan Nnastic, Schahram Dustdar, Deviceless edge computing: Extending serverless computing to the edge of the network, in: Proceedings of the 10th ACM International Systems and Storage Conference, SYSTOR '17, ACM, New York, NY, USA, 2017, p. 28:1.
[19] Node-RED, Node-RED. https://nodered.org/. (Online; accessed 25 July 2017).
[20] Alex Glikson, TRANSIT: Flexible pipeline for IoT data with Bluemix and Openwhisk. https://medium.com/openwhisk/transit-flexible-pipeline-for-iot-data-with-bluemix-and-openwhisk-4824cf20f1e0. (Online; accessed 25 July 2017).
[21] GRyCAP, SCAR Github repository. https://github.com/grycap/scar. (Online; accessed 25 July 2017).
[22] Theano, Theano. http://deeplearning.net/software/theano. (Online; accessed 25 July 2017).
[23] Joseph Redmon, Darknet. https://pjreddie.com/darknet/. (Online; accessed 25 July 2017).
[24] Erlang, Erlang. https://www.erlang.org/. (Online; accessed 25 July 2017).
[25] Jose Valim, Elixir. https://elixir-lang.org/. (Online; accessed 25 July 2017).
[26] ImageMagick Studio, Image Magick. https://www.imagemagick.org. (Online; accessed 25 July 2017).
[27] FFmpeg, FFmpeg. https://ffmpeg.org/. (Online; accessed 25 July 2017).
[28] Amazon, Amazon CloudWatch. https://aws.amazon.com/cloudwatch. (Online; accessed 25 July 2017).
[29] Virtuozzo, OpenVZ. https://openvz.org. (Online; accessed 25 July 2017).
[30] Canonical, LXC. https://linuxcontainers.org/. (Online; accessed 25 July 2017).
[31] Docker, Docker Hub. https://hub.docker.com/. (Online; accessed 25 July 2017).
[32] Jorge Gomes, udocker. https://github.com/indigo-dc/udocker. (Online; accessed 25 July 2017).
[33] Jorge Gomes, Isabel Campos, Emanuele Bagnaschi, Mario David, Luis Alves, Joao Martins, Joao Pina, Alvaro Lopez-Garcia, Pablo Orviz, Enabling rootless linux containers in multi-user environments: the udocker tool, 2017. arXiv preprint arXiv:1711.01758.
[34] D. Salomoni, I. Campos, L. Gaido, G. Donvito, P. Fuhrman, J. Marco, A. Lopez-Garcia, P. Orviz, I. Blanquer, G. Molto, M. Plociennik, M. Owsiak, M. Urbaniak, M. Hardt, A. Ceccanti, B. Wegh, J. Gomes, M. David, C. Aiftimiei, L. Dutka, S. Fiore, G. Aloisio, R. Barbera, R. Bruno, M. Fargetta, E. Giorgio, S. Reynaud, L. Schwarz, INDIGO-Datacloud: foundations and architectural description of a Platform as a Service oriented to scientific computing. Technical report, INDIGO-DataCloud, Mar 2016.
[35] Jorge Gomes, udocker documentation. https://github.com/indigo-dc/udocker/blob/udocker-fr/doc/user_manual.md. (Online; accessed 25 July 2017).
[36] JPiotr Roszatycki, Fakechroot. https://github.com/dex4er/fakechroot. (Online; accessed 25 July 2017).

[37] Amazon, Boto 3. http://boto3readthedocs.io. (Online; accessed 25 July 2017).
[38] GRyCAP, FFmpeg on AWS Lambda. https://github.com/grycap/scar/tree/master/examples/ffmpeg. (Online; accessed 25 July 2017).
[39] Amazon, AWS Lambda. How it works. http://docs.aws.amazon.com/es_es/lambda/latest/dg/lambda-introduction.html. (Online; accessed 25 July 2017).
[40] Amazon, Invoke. http://docs.aws.amazon.com/es_es/lambda/latest/dg/API_Invoke.html. (Online; accessed 25 July 2017).
[41] CentOS, Docker Hub: centos:7. https://hub.docker.com/_/centos/. (Online; accessed 25 July 2017).
[42] Bitnami, Docker Hub: bitnami/minideb. https://hub.docker.com/r/bitnami/minideb/. (Online; accessed 25 July 2017).
[43] GRyCAP, Docker Hub: grycap/jenkins:ubuntu14.04-python. https://hub.docker.com/r/grycap/jenkins. (Online; accessed 25 July 2017).
[44] Joseph Redmon, Darknet: Open source neural networks in C, 2013-2016. http://pjreddie.com/darknet/.
[45] Joseph Redmon, Ali Farhadi, Yolo9000: Better, faster, stronger, 2016. arXiv preprint arXiv:1612.08242.
[46] GRyCAP, Docker Hub: grycap/darknet. https://hub.docker.com/r/grycap/darknet/. (Online; accessed 25 July 2017).
[47] Amazon, Amazon Lambda pricing calculator. https://s3amazonaws.com/lambda-tools/pricing-calculator.html. (Online; accessed 25 July 2017).

Alfonso Pérez received B.Sc. + M.Sc. degrees in Computer Science and M.Sc. degree in Computer Engineering from the Universitat Politècnica de València (UPV), Spain, in 2011 and 2014, respectively. In 2011 he initiated his research career working in medical informatics projects related with clinical prediction models and computer interpretable guidelines. He has been member of the Grid and High Performance Computing research group (GRyCAP) at the Institute for Molecular Imaging (I3M) since 2015 where he began doing research work related with automatic cluster deployment and elasticity. In 2016 he started his Ph.D. in the field of Big Data and High Throughput Computing on Container-based and Serverless Infrastructures.

Germán Moltó received B.Sc. and Ph.D. degrees in Computer Science from the Universitat Politècnica de València (UPV), Spain, in 2002 and 2007, respectively. He has been a member of the Grid and High Performance Computing research group (GRyCAP) at the Institute for Molecular Imaging (I3M) since 2002. He is also an associate professor in the Department of Computer Systems and Computation (DSIC) at UPV. He has participated in several European projects and led national research projects in the area of cloud computing. His broad research interests are cloud computing and scientific computing.

Miguel Caballer obtained B.Sc., M.Sc., and Ph.D. degrees in Computer Science from the Universitat Politècnica de València (UPV), Spain, in 2000, 2012, and 2014, respectively. He has been a member of the Grid and High Performance Computing research group at the Institute for Molecular Imaging (I3M) since 2001. He has participated in several European and national research projects related to the application of parallel, grid, and cloud computing techniques to various areas of engineering. His other fields of interest include green computing.

Amanda Calatrava received the B.Sc., M.Sc. and Ph.D. degrees in computer science from the Universitat Politècnica de València (UPV) in 2010, 2012 and 2016, respectively. In 2011, she joined the Grid and High Performance Computing research group as a graduate under a collaboration fellowship while she worked on her Master's Thesis. She obtained her Master's Degree in February 2012, in which she specialized in grid and cloud computing. In November 2016, she obtained the Ph.D. degree whose work was focused in the field of virtual elastic clusters over hybrid cloud infrastructures. Currently, her research interests are focused in Virtualization, Cloud Computing and Scientific Computing.